Quick Start Guide

for Porting Applications for Win64

Table of Contents

1.0 Introduction

This document is meant to be a reference for application developers who are porting their application to the Win64 platform. The document first describes the different porting options available in the Win64 environment. Next, it lists the steps required for porting applications from Win32 to Win64. Finally, some of the most commonly encountered Win64 related errors and warnings are discussed.

To begin the porting efforts you will need access to the Win64 porting tools. The Microsoft platform SDK ships with a pre-beta release of an IA-64 compiler, which allows you to test compile your code for Win64. Information is available from the Microsoft Developer Network web site (http://msdn.microsoft.com/developer/sdk/platform.asp).

2.0 Choosing the Appropriate Porting Model

Win64 provides 4 different porting options for your application. The kind of option you choose will depend on the application needs.

1. 64-bit Full Port

   A 64-bit full port is the optimal scenario to take full advantage of IA-64 performance. A full 64-bit port should be considered if there is a current (or future) need for more than 2 GB of address space. Data and Code expansion are not an issue. A 64-bit full port requires you to do a full code clean, i.e., eliminate all of the warnings discussed in section 4.0. External modules, such as DLLs that are acquired from (or provided to) third parties, should consider a full 64-bit port. Also, 64-bit full port should be considered for applications where there is not an easy partitioning between the 32-bit and 64-bit sections.

2. Small Address Space with 64-bit Pointers

   This model should be considered if there is no need for a larger address space (2 GB is sufficient). Data size expansion is not an issue. For this model you can ignore most of the pointer truncation warnings. All addresses may safely be truncated into a 32-bit quantity (the upper 32 bits are 0).

3. Small Address Space with 32-bit Pointers

   This model should be considered if there is no need for a larger address space (2 GB is sufficient) and there are some concerns about the data size expansion. Again, over here you can ignore most of the pointer truncation warnings. 64-bit pointers are still required for interaction with OS.

4. Win32 Application

   In this option you do not need to do anything. This option should be chosen if the IA-32 performance is adequate and there is no need for address space > 2 GB.

In Win64 32-bit modules and 64-bit modules cannot co-exist in the same process. So applications planning on using a combination of 64-bit modules with 32-bit modules need to use RPC or COM/DCOM based Inter Process Communication mechanism to connect the IA-64 and IA-32 processes together. This option should be chosen if:

- There is easy partitioning between the 32-bit and 64-bit modules;
- Inter Process communication wrappers are easier to implement than a full 64-bit port; or
- There is a concern about the memory size expansion with pointer intensive data sets.

3.0 Steps for Porting Applications from Win32 to Win64

The following section lists the steps required for porting the application from Win32 to Win64 in a pre-silicon environment. The goal is to have the application ready to run on IA-64-based platforms running Win64 when they are available.

1. **Select the appropriate Porting Model.** Section 2.0 lists the options available for porting in the Win64 Programming Model
2. **Compile the application under Win32 environment.** Verify that the program works under the base platform. Take a note of the existing warnings and try to fix them. (These are low priority, the prime directive is successful removal of Win64 related warnings.)
3. **Backup the existing makefile(s) for the application.**
4. **Make the following changes to the makefile(s) for Win64.**

   - Remove /FD compiler flag:

     The /FD flag is generated when exporting makefile(s) from Visual Studio* and should be deleted.

   - Change the linker option which specifies the machine type as IX86 or I386 to IA64:

     machine:I386 → machine:IA64
     machine:IX86 or machine:x86 → machine:IA64

- Remove the /Gm compiler flag:

  The /Gm option controls minimal rebuild and is ignored by the IA-64 compiler.

- Add the following compiler flags to enable the most sensitive IA-64 related compiler warnings: -Wp64 and -W4

- If in step 1, you decided to use 32-bit pointer variables add the following compiler option -Ap32 (default is -Ap64).

- If in step 1, you decided to use a maximum of 4 GB (2^32) of virtual address space (Small Address Space) add the following compiler option -As32 (default is -As64).

5. **Clean the existing .obj files from the Win32 build and any existing Win64 builds.**

   nmake -f *makefilename* clean

6. **Run the modified Win64 makefile.**

   nmake -f *makefilename*

   You might want to output all the Win64 warnings that are produced to a text file.

   nmake -f makefilename > warnings.txt

   As changes are made to the source code these warnings will fluctuate. The warnings.txt file will provide you with a base reference.

7. **Make changes to the application code, one at a time, to fix the IA-64 related warnings.** The changes that are made should not impact the IA-32 functionality. Use code guards to protect and isolate changes : #ifdef _WIN64…#else…#endif

   Section 4.0 discusses some of the commonly encountered Win64 warnings and ways to fix them.

8. **Repeat step 7 for every Win64 related warning.**

9. **Execute regression tests on IA-32 to ensure that nothing was broken by the modifications.**

When production compilers, operating systems and hardware are available, the code can be re-compiled and regression tests executed on real hardware.

4.0 Handling Commonly Encountered Win64-related Errors and Warnings

Win64 uses the LLP64 (or P64) uniform data model in which pointers and long long are 64 bits, but int and long are 32 bits. To support this data model and to provide compatibility with the existing Win32 code, new data types have been defined in Win64. Some of them are fixed precision data types, for instance INT32 or INT64. Others change size depending on the architecture, for instance INT_PTR. The size will be either 32 bits or 64 bits depending on how you're compiling. For details refer to: http://msdn.microsoft.com/isapi/msdnlib.idc?theURL=/library/sdkdoc/buildapp/64bitwin_7zg3.htm

Most of the warnings that you encounter when compiling for Win64 are truncation-related warnings (conversion from INT64 to int).

A truncation related warning might look something like this:

C4311: Truncation warning. For example, "type cast": pointer truncation from "int*_ptr64" to "int."

There are two ways to fix truncation-related warnings:

- Cast the result to a proper data type
- Change the receiving variable to a scalable data type

Example:

1. **Casting:** Most of the non-pointer variables can be safely casted as long as the result is guaranteed to be less than the maximum value that can be held in the casted data type.

Pointer Difference

```
Before:  char *start, *end                    // start and end of buffer
         int x = end - start;                 // start/end will be 64 bits

After:   char *start, *end                    // start and end of buffer
         int x = (int)(end - start);          // cast the result
```

Here we know that the difference (end-start) is less than 4 GB, hence it can be safely casted.

2. **Changing Type:** By changing the receiving variables to a scalable data type the same source code would work for IA-32 as well as IA-64. Scaling can cause a rippling effect where scaling a result would require several other variables and/or functions to be scaled. This will generate more warnings, which need to be addressed.

Variables receiving pointers

```
Before:  char p;
         int j = &p;                          // address wil be 64 bits

After:   char p;
         INT_PTR j = &p;                      // scale the result
```

Passing parameters into functions

```
Before:  void func(int s);
         ..........
         char *p;
         func(p);                             // p will be 64 bits

After:   void func(INT_PTR s);                // scale the variable
         ..........
         char *p;
         func(p);                             // p will be 64 bits
```

A few of the constants used with the Win32 APIs have been modified for Win64; as a result, using the old constants will generate an error. The error might look something like this

error C2065: 'GWL_HINSTANCE' : undeclared identifier

The error can be fixed by replacing the old constant with the new constant.

In addition, four of the existing Win32 APIs that are used to set or get polymorphic window class data items have been changed.

Get/SetClassLong changed to Get/SetClassLongPtr

Get/SetWindowLong changed to Get/SetWindowLongPtr

For details refer to:
http://msdn.microsoft.com/isapi/msdnlib.idc?theURL=/library/sdkdoc/buildapp/64bitwin_9xo3.htm

Example:

Using Win32 API

Before: *LONG iVal = GetWindow**Long**(hWnd, **GWL_HINSTANCE**);*

After: *LONG_PTR iVal = GetWindow**LongPtr**(hWnd, **GWLP_HINSTANCE**);*

Using improper format specifiers in printf or wsprintf will generate warnings. The warnings might like something like this:

C4313: Calling the **printf** family of functions with conflicting conversion type specifiers and arguments.

Using the proper format specifiers will fix this warning

Example:

Before:	*printf("%p", int_value)*	*// this code expects %p to be // 32 bits and will fail on IA-64*
After:	*printf("%x",int_value)*	*// change to %x for IA-64 // to get desired result*
Before:	*printf("%x", ptr_value)*	*// %x is a 32 bit field and // will truncate the pointer*
After:	*printf("%p",ptr_value)*	*// change to %p to dsiplay // the full pointer value*

Visual C++ also supports the %I64 prefix for printing 64 bit values. The I64 prefix is a Microsoft specific extension and is not ANSI Compatible.

In addition there are certain things for which the compiler will not generate warnings, but you will have to look into for generating clean IA-64 code. These are placing the IA-64 code guards at the right places, using appropriate Win64 helper functions to convert from one type to another, handling data type alignments and using the appropriate typecasts for the ~ operator. For details refer to the following URLs:

- http://msdn.microsoft.com/isapi/msdnlib.idc?theURL=/library/sdkdoc/buildapp/64bitwin_410z.htm
- http://msdn.microsoft.com/library/sdkdoc/buildapp/mi-port64_9ius.htm

For more information

- http://developer.intel.com/vtune/cbts/ia64tuts/clean/index.htm

  An excellent tutorial, which explains the porting issues and guidelines that enable you to port your existing Win32 application to a Win64 application
- http://developer.intel.com/design/ia64/getsoftready/index.htm

  Getting your software ready for the IA-64 Architecture

1199/HB/KW/1K 283293-001